Intel® Itanium® Processor Family Error Handling Guide

April 2004

Document Number: 249278-003

Contents

Figures

Tables

Revision History

Date	Description	Date
-003	Updated links to related documents. Changes to reflect updated trademarks. Provided differentiation for machine checks vs. MCA references. Updated PAL, EFI, PMI, Data Poisoning and MCA definitions. Updated diagram for the Error Handling Flow. Removed references to IA-32 Operating Environment. Removed Chapter 6: OS Error Handling. Corrected Local and Global MCA references.	January 2004
-002	Added definitions for Domain, CMCI, and Corrected Error. Multiple clarifications to Chapter 2, Machine Check Architecture. Added sections on SAL/OS responsibilities and SAL issues with nested errors. Clarifications to Chapter 3, Processor Error Handling. Clarifications to Chapter 4, Added detail and new sections on the scope and handling of platform-corrected errors. Added detail to Error Record definition in Chapter 5, including new sections on required/optional error records. Added clarifications to Section 6.2, Identifying the Errant and Affected Threads. Added additional detail to Appendix A, Error Injection.	August 2001
-001	Initial release of the document.	January 2001

§

Introduction 1

1.1 Purpose

This Guide describes error handling on Intel® Itanium® architecture-based systems. It provides guidelines for firmware and operating systems to take advantage of the Machine Check Architecture of Itanium architecture-based systems. This Guide references the Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL) specifications and shows how firmware, platform design, and the operating system (OS) work together to address machine check aborts.

1.2 Target Audience

This document is intended for Itanium processor SAL developers, platform designers, and OS developers. Implementation-specific details of processors are not discussed here. The target audience is expected to be familiar with the PAL and SAL specifications.

1.3 Related Documents

This document refers to the following publications:

- *Itanium® Software Conventions and Runtime Architecture,* downloadable at http://developer.intel.com/design/itanium/downloads/245358.htm
- *Intel® Itanium® Architecture Software Developer's Manual.*, downloadable at http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm
- *Intel® Itanium® Processor Family System Abstraction Layer Specification,* downloadable at http://www.intel.com/design/itanium/downloads/245359.htm
- *Extensible Firmware Interface Specification,* downloadable at http://developer.intel.com/technology/efi/main_specification.htm
- *Intel® 460GX Chipset System Software Developer's Manual,* downloadable at http://www.intel.com/design/itanium/downloads/248704.htm

1.4 Terminology

ACPI
Advanced Configuration and Power Interface Specification.

AP
Application Processor. Any one of the processors not responsible for system initialization.

API
Application Programming Interface.

BERR#
Bus Error Signal. A platform error signal generated by a BERR# is potentially recoverable based on the OS error handling capabilities and platform log information. BERR# may also be driven by the processor on non-recoverable errors to notify other processors on the bus that an error occurred. This signal is maskable by the processor's psr.mc bit.

BINIT#
Bus Initialization Signal. An unmaskable system wide signal driven by the processor or platform bus to indicate a fatal machine check condition. This error clears the processor state and brings all processors to MCA.

BIOS
Basic Input/Output System. A collection of routines that includes Power On Self-test (POST), system configuration and a software layer between the OS and hardware. BIOS is written in IA32 instruction set.

BSP
Bootstrap Processor. The processor responsible for system initialization.

CMC
Corrected Machine Check from processor corrected errors.

CMCI
Corrected Machine Check Interrupt. An interrupt generated by processor hardware following a processor corrected or PAL-corrected error.

CPE
Corrected Platform Errors are the errors originating from platform-detected errors.

CPEI
Corrected Platform Error Interrupt. An interrupt generated by the platform following a platform-corrected error.

CPEV
Corrected Platform Error interrupt vector.

Data Poisoning
A state of data that is non-correctable and remains in this state in the storage media. (i.e. 2xECC or multi-bit errors in cache or memory).

Domain
Partitioning the system resources into separate entities with processor nodes, I/O nodes and memory. A domain is isolated from the hardware errors occurring on other domains, except for errors on common system hardware such as power supplies, interconnects, etc.

ECC
Error Correcting Code. This term refers to a code that is used to detect and correct data corruption for the storage media or data transmission on a bus. For this guide, the error correcting code described allows processor/platform hardware to automatically correct 1-bit errors (1xECC) and detect and signal multi-bit errors (2xECC).

EFI
The Extensible Firmware Interface is the interface between the OS and platform firmware. The interface consists of platform-related information, plus boot and runtime service calls that are available to the OS and its loader. This provides a standard environment for booting an OS and running pre-boot applications.

Error Categories:

Corrected Error
All errors of this type are either corrected by the processor (CMC), platform hardware (CPE), or SAL>PAL. Corrected errors are logged but may be overwritten by subsequent logs, so corrected logs may not always be correlated to OS event notifications. OS interaction is required for processor corrected error logging, but not error correction.

Recoverable Error
An uncorrected error occurred which had corrupted state, and the state information is known. Recoverable errors cannot be corrected by either the hardware or firmware. This type of error requires OS analysis and a corrective action to recover. System operation or state may be impacted.

Fatal /Non-Recoverable Error
An uncorrected error which occurred and resulted in a corrupted state, may not be known or contain adequate state information. This type of error cannot be corrected by the hardware, firmware, or the OS. The integrity of the system, including the I/O devices is not guaranteed and may require I/O device initialization and a system reboot to continue. Fatal errors may or may not have been contained within the processor or memory hierarchy. If the error is not contained, it must be reported as fatal.

GUID
A Globally Unique Identifier is a number of sufficient length to guarantee uniqueness when assigned to a member of a group. GUIDs on Itanium architecture are used to identify structures, procedures, and data in firmware.

Hard Fail Bus Response
Hard Failure. A processor system bus response used to indicate a transaction failure to the requesting agent of the transaction.

IA-32 Architecture
The 32-bit and 16-bit Intel architecture as described in the *Intel® Itanium® Architecture Software Developer's Manual.*

Itanium® Architecture
Intel's instruction set architecture with 64-bit addressing capabilities, new performance enhancing features, and support for IA-32 applications.

Itanium® Architecture-based Operating System
An operating system that can run Itanium architecture-based applications and optional legacy applications.

IPI
Inter-processor interrupt signaling using the local SAPIC within the processor.

MC Rendezvous Interrupt
An interrupt used to signal the OS to enter a rendezvous spin-loop in firmware, to quiesce the system during an MCA.

MCA
Machine Check Abort. MCAs are classified into two categories: Local and Global.

Local MCA
The local MCA is limited to the processor or the thread that encountered the internal error or a platform error and is not broadcast to other processors in the system. At any time, more than one processor in the system may experience a local MCA and handle it without notifying the other processors in the system. Local MCAs are signaled by either internal processor errors (multi-bit errors) or Hard Fail bus responses.

Global MCA
A global MCA may result in a system wide broadcast of an error condition, depending on the platform implementation. During a global MCA, all the processors in the same system domain will be notified of an MCA event. Global MCAs are signaled via the BERR# and BINIT# bus signals and are asynchronous to the instruction stream and to the transactions on the system bus.

Min-State Save Area
Area registered by SAL with PAL for saving minimal processor state during machine check and INIT processing. See the *Intel® Itanium® Architecture Software Developer's Manual* for details.

Monarch Processor
The processor elected by some implementations of SAL or the OS to coordinate the platform error handling when multiple, simultaneous machine check events occur in a multiprocessor system.

MP
Multiprocessor.

Node
A node may consist of processors, memory and in some cases I/O devices. A system may contain multiple nodes, such as multiple processor systems.

NVM
Non-volatile memory.

OS
Operating System.

PAL
The Processor Abstraction Layer of firmware which abstracts processor features that are implementation dependent.

PMI
Platform Management Interrupts (PMI) provide an operating system-independent interrupt mechanism to support OEM and vendor-specific hardware events. See the *Intel® Itanium® Architecture Software Developer's Manual* for details.

SAL
The System Abstraction Layer of firmware which abstracts system features that are implementation dependent.

SAPIC
Streamlined Advanced Programmable Interrupt Controller is the high performance interrupt mechanism used on Itanium architecture-based systems. The **Local SAPIC** resides within the processor and accepts interrupts sent on the system bus. The **IOxAPIC** resides in the I/O subsystem and provides the mechanism by which I/O devices inject interrupts into the system.

TLB
Translation Lookaside Buffer.

Wakeup Interrupt
Interrupt sent by the OS to wake up the APs from the SAL_MC_RENDEZVOUS spin loop. This interrupt vector is registered by the OS with SAL.

§

Machine Check Architecture 2

2.1 Overview

System error detection, containment, and recovery are critical elements of a highly reliable and fault-tolerant computing environment. While error detection is mostly accomplished through hardware mechanisms, system software plays a role in containment and recovery. The degree to which this error handling is effective in maintaining system integrity depends upon coordination and cooperation between the system CPUs, platform hardware fabric, and system software.

The Machine Check Architecture provides error handling features for high reliability, availability, and serviceability. Error containment is the highest priority, followed by error correction without program interruption, and the recording of error information.

This chapter provides an overview of the Itanium processor family firmware and describes the machine check error handling model. A discussion on the scope and classification of errors will be provided. Finally, topics that deviate from the normal error handling model are covered.

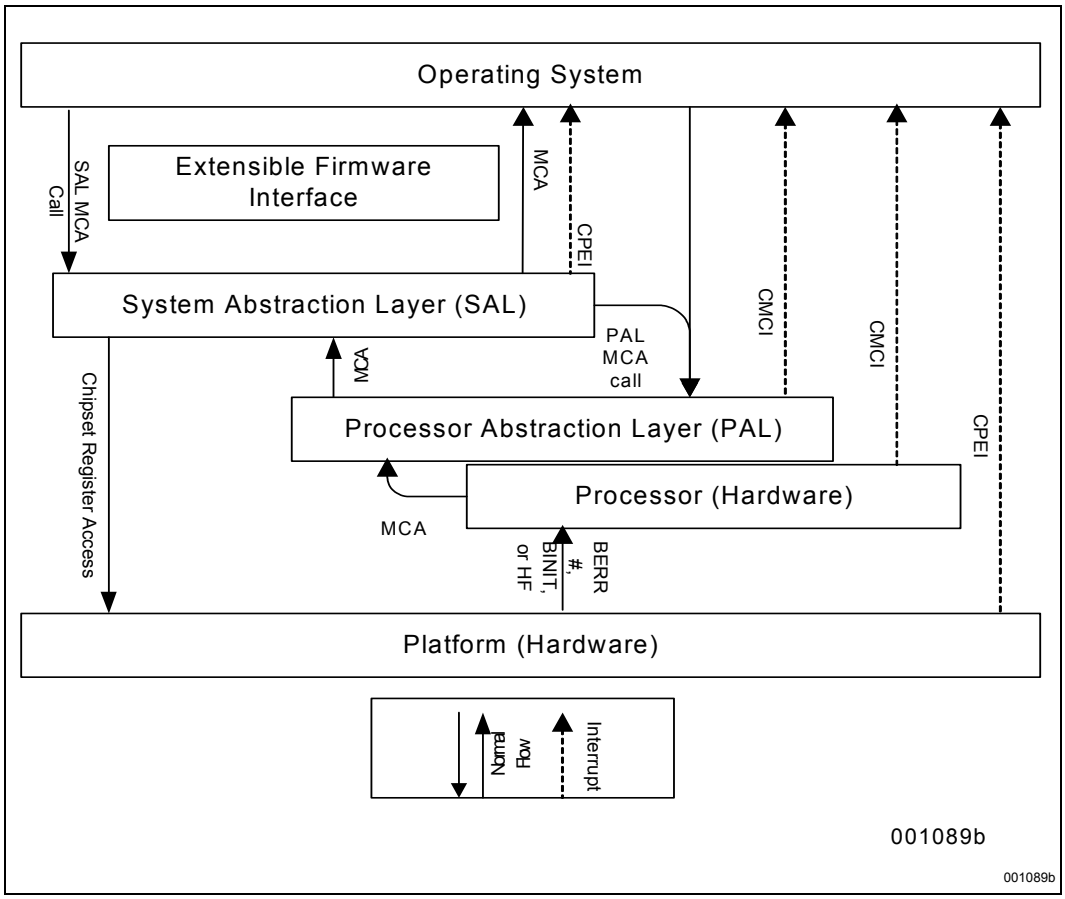
2.2 Itanium® Processor Family Firmware Model

The Itanium architecture defines three firmware layers: the Processor Abstraction Layer (PAL), the System Abstraction Layer (SAL), and the Extensible Firmware Interface (EFI) layer.

- PAL encapsulates those processor functions that are likely to change based on implementation so that SAL firmware and operating system software can maintain a consistent view of the processor. These include non-performance critical functions such as processor initialization, configuration and error handling.
- SAL is the platform-specific firmware component provided by OEMs and firmware vendors. SAL and EFI isolate the OS and other higher level software from implementation differences in the platform.
- EFI is the platform-binding specification layer that provides a legacy-free API interface to the OS loader and a means of supporting value-added platform features.

PAL, SAL, and the OS work together to handle machine check aborts, processor-corrected errors, and platform-corrected errors. Figure 2-1 provides an overview of how the firmware and OS interact for machine check handling.

Figure 2-1. Itanium® Processor Family Firmware Machine Check Handling Model



2.3 Machine Check Error Handling Model

The Itanium architecture error handling model consists of different software components that work in close cooperation to handle different error conditions. PAL, SAL, and the OS have error handling components, which are tightly coupled through a well defined interface.

System errors may be handled by any of the following components:

1. Processor Hardware
2. Platform Hardware
3. PAL
4. SAL
5. Operating System

Hardware Error Handling: When the processor or platform hardware corrects an error, a notification of the corrected event is signaled through a CMCI for processor-corrected errors or a CPEI for platform-corrected errors. The OS also has a choice to disable this automatic interrupt notification and can periodically poll the firmware to collect corrected error events. The OS can

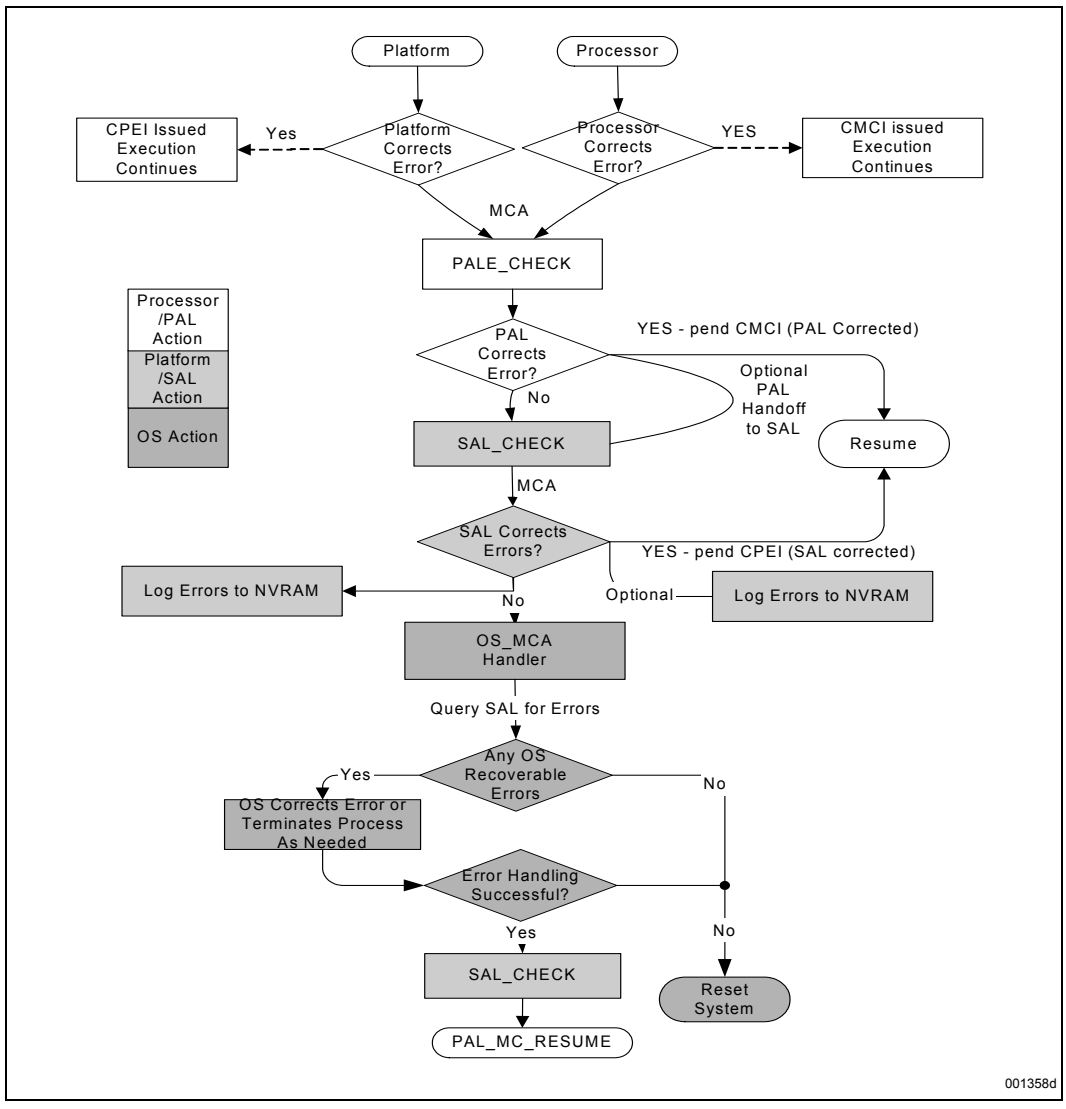
also chose to mask interrupts using external interrupt vector numbers for Corrected Machine Check events, (for more information about masking interrupts using CMCV.m, refer to the *Intel® Itanium® Architecture Software Developer's Manual, Volume 2).*

Firmware Error Handling: When the processor or platform hardware detects an error that is not correctable directly by hardware, an MCA event is triggered. The MCA event will pass control to the firmware. The PAL and SAL firmware will correct any errors that they are capable of. Errors that are corrected by firmware are recorded, and the control is returned back to the interrupted context. These corrected errors require no OS intervention for error handling. If an error is not correctable by firmware, control is then passed to the OS.

Operating System Error Handling: When an error is not corrected by the hardware or firmware layers, the control is transferred to the OS. The OS will correct any errors that it can and will either return to the interrupted context, switch to a new context, or reset the system.

Figure 2-2 illustrates the high level view of the machine check error handling flow. For more details, refer to the *Itanium® Processor Family System Abstraction Layer (SAL) Specification.*

Figure 2-2. Machine Check Error Handling Flow



2.4 MCA Scope

Both global MCA and local MCAs are implementation concepts and not reflected in the PAL/SAL architectural state hand-offs or error logs. The scope of a global MCA depends on the platform implementation, with the exception of BINIT# which is always a system-wide event.

Local MCA: The scope of a local MCA is limited to the processor that encountered the internal error or a platform error. This local MCA will not be broadcast to other processors in the system. At any time, more than one processor in the system may experience a local MCA and handle it without notifying the other processors in the system. In certain cases, the firmware may rendezvous other processors in the system for coordinating the error handling. Local MCAs are signaled by either internal processor errors, multi-bit errors or Hard Fail bus responses.

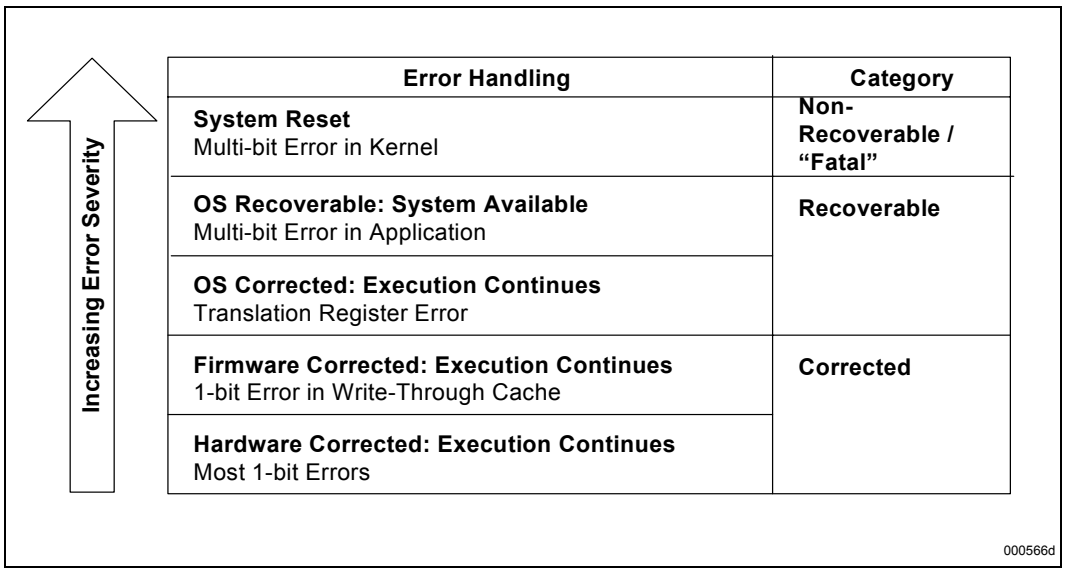
Global MCA: A global MCA may result in a system-wide broadcast of an error condition, depending on the platform implementation. During a global MCA event, all the processors in the same system domain will be notified of an MCA. In a global MCA, all the processors in the domain will enter their respective MCA handlers and start processing the error event. The system firmware and OS layers will each coordinate the handling of the error in the MP environment. A platform may promote BERR# to become a system wide event, but BINIT# is always system wide.

MCA error scope is not directly visible to the OS through any hand-off state information. It is documented here in an effort to describe the effects of errors in an MP system.

2.5 Error Types

Errors are classified into five different categories based on the severity and the scope of errors as shown in Figure 2-3.

Figure 2-3. Error Types and Severity



2.5.1 Corrected Error with CMCI/CPEI (Hardware Corrected)

All errors of this type are either corrected by the processor or platform hardware and have no impact on the currently executing process. Firmware is not involved in error correction for these types of events. The OS is notified of this event through a signaling mechanism (CMCI/CPEI) in order to facilitate error record retrieval. An OS can configure the system to disable the notification of the corrected error events, in which case it shall poll for these events through the SAL_GET_STATE_INFO procedure. There may be some loss of corrected error information if the OS polling frequency of SAL_GET_STATE_INFO is not adequate.

Examples of this type of error are a correctable 1-bit ECC error in the processor cache or a correctable 1-bit ECC error on the system bus.

Figure 2-4 represents a machine check error flow in which the errors were corrected without involving the OS.

2.5.2 Corrected Error with Local MCA (Firmware Corrected)

This type of error is not corrected by the processor or platform hardware and must be corrected by firmware. On detecting such an error, the processor signals a local MCA, forcing control transfer to the firmware. Processor-detected errors of this type are corrected by PAL, whereas platform-detected errors of this type are corrected by SAL. The firmware handlers correct the error and resume the execution of the interrupted context. When the error is corrected by the firmware layers, the corrected event is signaled to the OS as a CMC or CPE interrupt if the OS has enabled these, otherwise the OS needs to poll for this information.

An example of this type of error is an error that occurs in a write-through cache data structure containing unmodified data. The firmware invalidates the affected lines in the structure and returns execution to the interrupted process.

2.5.3 Recoverable Error with MCA

Recoverable errors can be due to a local MCA or global MCA as described below.

2.5.3.1 Recoverable with Local MCA

Recoverable errors of the local MCA type cannot be completely corrected by either the hardware or firmware. This type of error requires OS analysis of the error. The control and handling of this error is left to the OS. Recovery is not always possible, depending on the capability of the OS and the error record information provided to it by the firmware. When an error is not recoverable, the system should be rebooted to return it to a safe state.

Recoverable errors should have the following characteristics:

1. The error is contained (i.e. it has not been saved in persistent storage).
2. Critical system state is intact.
3. The physical address that caused the error and the instruction pointer of the offending instruction are captured.
4. The process and the system are continuable.

An example of a recoverable error with a local MCA is an error that returns incorrect data to a processor register. If the OS can identify the offending process from the error information logged, it can recover by terminating the process that needed to consume this data. A platform may also signal a recoverable error detected by the platform components through an appropriate signaling mechanism (e.g.Hard Failure bus responses, multi-bit errors).

2.5.3.2 Recoverable with Global MCA

These type of errors are similar to recoverable errors with local MCA except that these errors are broadcast to all the processors in the same system domain via a signaling mechanism (e.g. BERR# pin assertion). Refer to Section 4.4. On detecting a global error event condition, each of the processors enters its local MCA handler to perform error handling via SAL, with a subsequent hand-off to the OS. The eventual control and handling of these errors is left to the OS_MCA. Once the OS_MCA handler determines the severity of the errors, it examines the error logs in SAL_GET_STATE_INFO to determine the correct recovery method. Once recovery is confirmed, the logs are deleted using SAL_CLEAR_STATE_INFO.

An example of a global error condition is a platform error condition asserting BERR# pin on all processors, assuming that the chipset has the ability to route and drive BERR# signals to all the processors.

2.5.4 Fatal Error with Global MCA

This type of error cannot be corrected by the processor, platform, firmware, or the OS. The system must be rebooted. This type of error is broadcast to the system via a global event notification mechanism (i.e. BINIT#). Please refer to Section 4.4. On detecting a global error event condition, all the processors enter their MCA handlers.

After a BINIT# signal, the first bus transaction must be the fetch to the MCA handler. The BINIT# error condition requires all the processors to handle the BINIT# reset. The BINIT# signal is not maskable by the processor's psr.mc bit. BINIT# (Bus Initialization signal) invalidates the state of outstanding memory and bus transactions.

An example of a BINIT# error condition is processor time-out expiration. This occurs when the processor has not retired any instructions after a specified watchdog time out period due to a system deadlock. Such an error will cause a BINIT# system reboot when enabled. A platform can also signal this error type through a BINIT# assertion.

2.6 Software Handling

The Itanium architecture requires PAL, SAL, and the OS to share the responsibility for machine check handling. The responsibilities of each of these components are described below.

2.6.1 PAL Responsibilities

Machine check abort events initially go to PAL for handling. PAL has the following responsibilities when receiving a machine check:

- Save the processor state in min-state save area of memory registered by SAL.
- Attempt to contain the error by requesting a rendezvous for all processors sharing a memory subsystem or all processors in the system if necessary. Refer to section 13.3.1.1 in *Intel® Itanium® Architecture Software Developer's Manual*.

- Attempt to correct the error.
- Hand off control to SAL for further processing and logging.
- Return the processor error information when the SAL requests it using the PAL_MC_ERROR_INFO procedure.
- Return to the interrupted context by restoring the state of the processor when PAL_MC_RESUME is called or when PAL corrects the error.

2.6.2 SAL Responsibilities

The responsibilities of the SAL may be categorized into initialization and runtime responsibilities.

2.6.2.1 SAL Responsibilities During System Initialization

The SAL has the following responsibilities during system initialization to set up the processor and the platform for appropriate MCA signalling:

- Invoke the PAL_BUS_SET_FEATURES procedure to enable BERR# and BINIT# sampling and to enable detection of errors on the data, address, request, and response signals on the system bus.
- Optionally, invoke the PAL_PROC_SET_FEATURES procedure to specify CMC, MCA, and BERR# promotion. Refer to Section 3.4.2 for more details. CMC promotion forces hardware corrected CMCs through the PAL and SAL layers.
- Program the chipsets on the platform to enable ECC generation on the data delivered by the chipset to the processor on the system bus and to enable error correction and detection logic on the data received by the chipset.
- Request virtual address registration for the platform hardware that the SAL needs to access during machine check processing, (e.g. chipset registers, NVM, etc.).
- The SAL may log MCA events that occur during the boot prior to the registration of the OS_MCA layer. However, it must preserve any unconsumed MCA records from the previous boot session as these may not have been recorded by the OS in persistent storage.

In general, the SAL should take maximum advantage of the error correction and detection capabilities provided by the platform hardware.

2.6.2.2 SAL Responsibilities During Machine Check Aborts

The SAL has the following responsibilities during machine checks:

- Attempt to rendezvous other processors in the system if requested to by PAL.
- Process MCA handling after hand-off from PAL.
- Retrieve the processor error record information from PAL_MC_ERROR_INFO for logging.
- Issue a PAL_MC_CLEAR_LOG request to clear the log and to enable further error logging.
- Initiate processor rendezvous, if the error situation warrants one.
- Elect a monarch processor if multiple processors enter the SAL MCA handler at the same time to coordinate error handling.
- Retrieve platform state for the MCA.

- Attempt to correct platform errors. If the error is not corrected and the OS has specified the "always rendezvous" flag through SAL_MC_SET_PARAMS, SAL will rendezvous the processors.
- Log the uncorrected error information to NVM and then hand off control to the OS_MCA handler. If the OS_MCA handler is absent or corrupted, then the SAL will either reset the system or take OEM-specific actions.
- On return from OS_MCA, if the error was corrected, return to the interrupted context through the PAL_MC_RESUME procedure. If the error was not corrected, the SAL may reset the system.

2.6.3 Operating System Responsibilities

The OS depends on SAL to interact with PAL to get information about machine check errors for further handling. The responsibilities of OS machine check handler may be categorized into initialization and runtime responsibilities.

2.6.3.1 Operating System Responsibilities During OS Initialization

To minimize the boot time, the following steps are required to be handled by the Bootstrap processor:

- Register spin loop/Rendezvous and Wakeup Request Interrupt Vectors.
- Specify the options during MCA processing (rendezvous always for MP systems, MCA to BINIT# escalation, etc.) by invoking the SAL_MC_SET_PARAMS procedure.
- Register an OS_MCA handler entry point by invoking the SAL_SET_VECTORS procedure.
- Initialize the CMCV register to enable CMC interrupt on the processor and install a handler for the interrupt. This is not required if the OS chooses to poll for corrected processor errors.
- Initialize the Corrected Platform Error Interrupt vectors (CPEI) in the I/O SAPIC. The details of the interrupt line on which CPEI is signaled is provided by the SAL in the *Platform Interrupt Source Structure* within the ACPI tables. The OS must also register the CPEV with the SAL using the SAL_MC_SET_PARAMS procedure. These steps are not required if the OS chooses to poll for corrected platform errors.
- Invoke the SAL_GET_STATE_INFO procedure on the bootstrap processor with argument types of CMC, CPE, and MCA to retrieve any unconsumed error records from the previous boot. Then, invoke the SAL_CLEAR_STATE_INFO procedure to mark such records as consumed, thereby freeing up the NVM storage for future logging of uncorrected errors.
- If an OS supports a mechanism to have an OEM call-back function for MCAs, it should set up this call-back function if required by the platform. The interface for such a function is OS specific.

The OS should then enable maskable interrupts on all the processors in the system. Machine checks would already have been unmasked when the OS gains control from EFI.

2.6.3.2 Operating System Responsibilities During a CMC at Runtime

On receipt of the CMCI, the OS should invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures with the CMC argument type on the processor on which the CMCI was signaled. This permits the SAL to mark the error record as consumed and free up

memory resources for future SAL use. The SAL_GET_STATE_INFO call must be invoked repeatedly until SAL returns a “No Information Available” status to the OS. If polling is employed, these procedures should be called periodically.

2.6.3.3 Operating System Responsibilities During a CPE at Runtime

On receipt of the CPEI interrupt, invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures with the CPE argument type on the processor on which the CPEI was signaled. This permits SAL to mark the error record as consumed and free up memory resources for future SAL use. The SAL_GET_STATE_INFO must be invoked repeatedly until the SAL returns a “No Information Available” status to the OS. If polling is employed, these procedures should be called periodically. Refer to Section 4.3.1 for additional information.

2.6.3.4 Operating System Responsibilities During an MCA at Runtime

The OS has the following responsibilities during machine checks:

- Elect a monarch processor, if multiple processors enter OS_MCA at the same time to coordinate error handling.
- Retrieve error records from SAL by making repeated calls to SAL_GET_STATE_INFO.
- Recover from the error if possible. If the OS cannot continue, it will return to the SAL and request a reboot.
- Call back the registered OEM machine check handler, if any.
- If the OS cannot continue further, return to the SAL, requesting a reboot. The SAL implementation will log the current MCA event to NVM and provide the MCA event information during the next reboot so that the OS may record it subsequently in persistent storage.
- If the error has already been corrected (as indicated by the ERR_SEVERITY field in the Record Header), or if the OS can continue, it should mark the MCA error record as consumed by invoking the SAL_CLEAR_STATE_INFO procedure.
- At the end of machine check processing, wake up all slave processors from the SAL rendezvous by sending the wake-up interrupt. If any processors go through INIT during the rendezvous, set up further processing steps as part of the OS_INIT procedure.
- On all the processors that entered the OS_MCA handler, the OS_MCA handler can specify a return to the interrupted context or branch to a new context by modifying the processor state that the OS provides to the SAL MCA handler.

2.6.3.5 Operating System Responsibilities During a Rendezvous Interrupt at Runtime

The OS has the following responsibilities during rendezvous interrupts:

- Identify the processors to be awakened at the end of OS_MCA processing, using an implementation-specific structure. This variable may be used by the OS to identify the processors that need to be woken up at the end of OS_MCA processing. The OS implementations may also use this variable to distinguish between a rendezvous interrupt and an INIT.
- Invoke the SAL_MC_RENDEZ procedure.
- On return from SAL_MC_RENDEZ at the end of MCA event processing, clear the interrupt pending bits within the processor for the rendezvous and wake-up interrupt vectors.
- Resume the interrupted context.

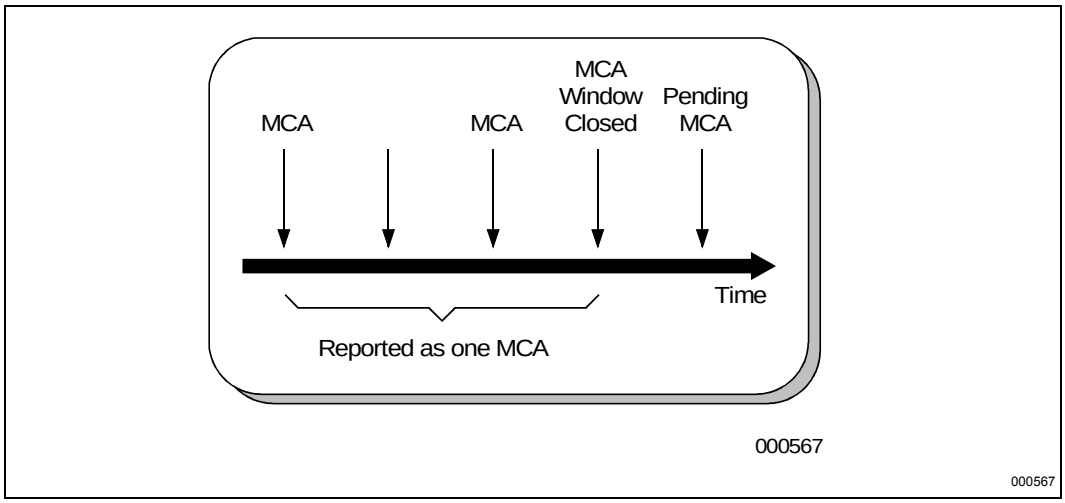
2.7 Multiple Errors

All machine check errors detected within a window before the processor masks machine check detection hardware (PSR.mc) are lumped together as a single MCA condition, locally visible on that processor only. This is shown in Figure 2-4.

Multiple MCA events within a detection window on a particular processor may be reported as:

1. A single error, if the same error is detected multiple times in the same structure (cache, TLB or system bus structural units).
2. A single error with an overflow indicator, if multiple unique errors are detected in the same structure it may be the result of shared logging and may be aggregated in a single overflow message.
3. Multiple unique errors in different structures.

Figure 2-4. Multiple MCA Events



Nested MCA: A nested MCA is an MCA that occurs after the MCA detection window is closed. All further MCAs occurring after the detection window are held pending and may be unrecoverable. The Machine Check Architecture allows for multiple nested MCAs to occur on each processor, but only one MCA may be handled at a time. Note that errors detected and corrected by hardware trigger the optional CMCI or CPEI event and are not considered to be MCAs or nested MCAs.

Multiprocessor System: Error handling may depend on the number of processors in a system. In an MP environment, because of the possibility of a global MCA error or simultaneous local MCAs on multiple processors, firmware and OS_MCA handlers must perform synchronization during error handling. The SAL firmware may perform a rendezvous of the processors based on the error encountered or may be configured by the OS to always rendezvous using the SAL_MC_SET_PARAMS procedure. Likewise, the OS may perform its own rendezvous of the processors based on the error encountered if not already done by the firmware.

2.7.1 SAL Issues Related to Nested Errors

:*Reentrancy*: The *Itanium® Processor Family System Abstraction Layer Specification* requires that the SAL procedures invoked during an MCA be made reentrant. This permits a SAL invocation for CMC or CPE to be interrupted by SAL procedure calls to retrieve MCA information. Reentrancy becomes important as operating systems develop strategies to service simultaneous MCAs in the

error handling flow to allow building and retrieval of error records on the same processor. For the environments where the OS does not attempt recovery from MCA, the minimum SAL implementation requirements are that:

- The MCA error information is provided to the OS_MCA layer.
- The MCA error record is logged to the NVM.

To simplify SAL implementation, it is strongly recommended that SAL process all MCAs by handing off to the OS as soon as possible to prevent some OSes from experiencing time-outs and potentially crashing the system. The SAL may maintain a variable in the SAL data area that indicates whether SAL, on one of the processors, is already handling an MCA. If so, MCA processing on other processors will wait within the SAL MCA handler until the current MCA is processed. This situation may arise when local MCAs are experienced on multiple processors. The SAL code must also implement methods of detecting which processors have already reached the SAL MCA handler and avoid steps to rendezvous such processors (using MC_rendezvous interrupt or INIT).

While SAL firmware is handling the MCA event, further machine checks are masked by the PSR.mc bit in the processor status register (PSR). This does not, however, mask a subsequent BINIT# event. SAL firmware must be designed to allow for a BINIT# occurring in the middle of a machine check event processing and must not wait on any semaphores that may have been held by the earlier MCA event. At a minimum, the SAL must log the BINIT# error to NVM and provide the error information pertaining to the BINIT# to the OS_MCA layer.

Indexed I/O Accesses: Even if the re-entrancy issue is solved by careful SAL implementation, there are atomicity issues due to legacy hardware that requires multiple accesses. A number of I/O devices are accessed using indexed accesses (e.g. PCI configuration space). To access a register within the PCI configuration space, an index needs to be written followed by a read or write to the data port (e.g. I/O ports CF8, CFC). It is possible for an MCA to occur while the SAL is in the middle of the SAL_PCI_CONFIG_READ/WRITE procedure. The index may have been written but the data may not have been read/written. If the SAL MCA handler needs to access the PCI configuration space, it must first save the index, access the areas of interest and then restore the index.

The combination of restricting processing to one MCA at a time, rendezvousing all the processors on an MCA, and the save/restore of the index will permit the SAL to work around indexed I/O issues. The SAL must also ensure that other layers within SAL, such as SAL_PMI, do not access the PCI configuration space. This is solvable by maintaining a SAL data variable reflecting current usage (0=Unused; 1=OS; 2=PMI; 3=MCA) and establishing rules for pre-emption. The CMPXCHG instruction may be used for atomic updates to the variable.

The PCI configuration space is a resource abstracted by SAL and it is possible to solve the indexed I/O problem for this resource. There are no solutions if multiple software layers access indexed I/O resources without any coordination. Hence, the SAL code handling the MCA event shall not use other indexed I/O resources such as the Real-Time Clock NVM if this resource is also used by the firmware during runtime calls by the OS.

Memory Attribute Aliasing: The SAL code and data areas will be accessed both during machine check processing and during normal OS execution. The normal execution mode during the MCA path is uncachable (UC) using the firmware image (around 4GB), with address translation disabled. The normal execution mode while the OS retrieves the CMC and CPE error records is cachable and writeback (WB) using the copy of SAL in memory, with address translation enabled. Both these contexts would need to access common data variables. The SAL implementation must be careful to avoid accessing the same location with differing memory attributes (UC vs. WB), otherwise additional MCAs may surface due to memory attribute aliasing.

2.8 Expected MCA Usage Model

In addition to the error handling model described in Section 2.3, "Machine Check Error Handling Model", the Machine Check Architecture provides an MC Expected (MCE) configuration option for platform/software testing purpose. This machine check expected option is enabled or disabled through the PAL_MC_EXPECTED procedure. When this option is set, the PAL machine check handler will deviate from its normal handling and will not attempt to perform error handling other than log generation (hardware error recovery action is not affected), but hands off control to SAL directly. The Machine Check Architecture does not restrict the usage of the MCE option, but it is intended to be used for software diagnostics only.

2.9 Machine Checks During IA-32 Instruction Execution

IA-32 instructions can be executed in either the Itanium system environment or the IA-32 system environment. When operating in the Itanium system environment, the processor offers the comprehensive machine check model described in Section 2.3, "Machine Check Error Handling Model". For more details on machine check handling in Itanium system environment, please consult the *Intel® Itanium® Architecture Software Developer's Manual,* Vol. 2: System Architecture.

The Itanium processor family supports Intel® Pentium® processor compatible machine checks in the IA-32 System Environment. For details on Machine Check Architecture on IA-32 processors, please consult the *Intel® Itanium® Architecture Software Developer's Manual*.

§

Processor Error Handling 3

On detecting an internal error, the processor asserts a machine check. Processor-corrected errors are signaled directly to the OS using CMCI by default. If the error is containable, the processor branches to PALE_CHECK to begin the error handling flow. If the error is not containable, BINIT# is asserted to reset the platform.

3.1 Processor Errors

Machine check errors are reported using five different structures. At any point in time, a processor may encounter an MCA or CMC event due to errors reported in one or more of the following structures:

1. Processor Cache
2. Processor TLB
3. System Bus
4. Processor Register File
5. Processor Microarchitecture

Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for detailed processor error reporting information regarding PAL_MC_ERROR_INFO. An overview of the types of processor machine check errors is presented in the following sections.

3.1.1 Processor Cache Check

Itanium architecture implementations may have several levels of processor cache. An implementation may organize a level of cache as separate instruction and data caches or as a unified cache. To make the error information independent of the processor's cache implementation, a processor will report error information in an architecturally-defined manner for recovery and logging.

PAL_MC_ERROR_INFO may return the following information when a cache error occurs:

1. Instruction or data/unified cache failure identification.
2. Data or tag failure identification.
3. Type of operation that caused the failure.
4. The cache way and level of failed location.
5. Index of the failed cache line.
6. Physical address that generated the machine check.

3.1.2 Processor TLB Check

Itanium architecture implementations may have several levels of processor cache translation look-aside buffers (TLBs). An implementation may choose to have separate instruction and data TLBs or a unified TLB.

PAL_MC_ERROR_INFO may return the following information when a TLB error occurs:[1]

1. Translation register (TR) or the translation cache (TC) error identification.
2. Indication of whether the error occurred in an instruction or data/unified TLB structure.
3. Type of operation that caused the TLB MCA.
4. Level of the TLB where the error was encountered.
5. Slot number of the TR that experienced the MCA.
6. Physical address that generated the machine check.

3.1.3 System Bus Check

Itanium architecture implementations will report a bus machine check for system bus transaction errors or system bus agents reporting a global bus error.

PAL_MC_ERROR_INFO may return the following information when a bus error occurs:[2]

1. Size of the transaction that caused the machine check.
2. Indication of whether this machine check was due to an internal processor error or due to an external bus notification.
3. The type of bus transaction that generated the machine check.
4. Identification of the requester and responder of the bus transaction that generated the machine check.

3.1.4 Processor Register File Check

Itanium processor implementations have large register files, which may be protected to detect errors. Errors encountered on protected register files will be returned in the register file check.

PAL_MC_ERROR_INFO may return the following information when a register error occurs:[3]

1. Register File ID and register number for the failure.
2. Operation that generated the register file error.

3.1.5 Processor Microarchitecture Check

Itanium processor implementations have many internal arrays and structures that may not be architecturally defined yet may still be designed to detect errors. Any errors detected in architecturally undefined structures are reported using the microarchitecture check. These error conditions may not be recoverable by OS software but may be logged for serviceability.

PAL_MC_ERROR_INFO may return the following information when a microarchitecture error occurs:[4]

1. Structure ID, array ID, way and level where the error occurred.
2. Operation that triggered the error.

1. For further details on the TLB error info, please refer to *Intel® Itanium® Architecture Software Developer's Manual*.
2. For further details on the bus error info, please refer to *Intel® Itanium® Architecture Software Developer's Manual*.
3. For further details on the register file error info, please refer to *Intel® Itanium® Architecture Software Developer's Manual*.
4. For further details on the microarchitectural check information., please refer to *Intel® Itanium® Architecture Software Developer's Manual*.

3.2 Processor Error Correlation

SAL must call PAL_MC_ERROR_INFO multiple times to retrieve all of the information associated with a machine check event. SAL calls PAL_MC_ERROR_INFO to get the error severity within the Processor State Parameter (PSP) and error map information through Processor Error Map. Subsequent calls are made to obtain detailed error information. The PSP and the processor error map values returned by the PAL have a global summary of the error, which enable SAL to identify and make subsequent PAL calls to get detailed error information for each structure.

Note: As defined in the SAL specification, SAL returns processor error information for the processor on which the SAL_GET_STATE_INFO call is made. To get the error information for all processors, multiple SAL calls must be made on each processor.

3.3 Processor CMC Signaling

Corrected machine check events on a processor may be signaled using two different control paths:

1. Hardware Corrected Processor Error
2. Firmware Corrected Processor Error

A machine check error corrected either by processor hardware or firmware translates into a CMC condition with eventual notification to the OS. The notification of the CMC condition to the OS is only necessary for recording the error information. The OS can use this information for system management. For the processor hardware or firmware to deliver the CMC interrupt to the OS, the CMC interrupt must be enabled on each of the processors with CMC vector initialized in the processor's CMCV register.

On a processor-corrected error, a CMC event can be transferred to the OS by two different methods. The OS may either initialize the processor to generate an interrupt (CMCI) for automatic signaling of a CMC, or the OS can periodically poll the CMC condition through SAL_GET_STATE_INFO. The OS can choose any low priority interrupt vector for this purpose by programming the processor's CMCV register.

If the OS chooses to use polling for corrected processor error events, it must periodically call SAL_GET_STATE_INFO with argument type of CMC on each of the processors in the system to check for the validity of CMC events and error records. The polling frequency is implementation-specific, and should be frequent enough to prevent loss of current data while providing space for additional logging.

3.4 Processor MCA Signaling

A machine check abort condition may be due to a processor error condition or an externally generated *asynchronous* platform BINIT# / BERR# signal or a *synchronous* Hard Failure bus response/forced multi-bit errors on the system bus. The processor detecting the MCA or a platform chipset component may assert a signal on the BINIT# pin. The platform may assert the BERR# signal to the processor. An MCA due to observing a BERR# assertion has no effect on the processor state, so it may be possible to resume execution of the interrupted context if the error is contained and corrected.

However, an MCA due to a BINIT# assertion will reset all outstanding transactions in the processor memory/bus queues, causing the processor to lose state information. Thus BINIT# does not allow the system to recover.

A processor can assert different signals to communicate an error condition to the external platform components. When an MCA is localized to a processor, no external signalling will be visible in the platform.

3.4.1 Error Masking

System software may disable MCAs or corrected machine check interrupts by masking these conditions through the PSR. These capabilities are highlighted in Table 3-1.

Table 3-1. Processor Machine Check Event Masking

Processor Register	Field	Event	Description
Processor Status Register (PSR)	PSR.mc	MCA	Mask or unmask machine check aborts on the processor. However, delivery of MCA caused by BINIT# is not maskable.
CMC Interrupt Vector Register	CMCV.m	CMCI	Mask or deliver the CMC interrupt.

In general, it is not necessary for either SAL or the OS to manipulate the PSR.mc bit. On taking an MCA, another MCA is automatically masked by the processor hardware and restored when SAL or the OS returns to the interrupted context through PAL_MC_RESUME.

The OS needs explicit enabling or disabling of MCA signaling to handle special situations. A good example of this is when the OS wants to bring the system to a rendezvous state when multiple MCAs are detected. The OS or SAL could enable subsequent MCAs after the error record for the current MCA is logged to a nonvolatile storage area.

3.4.2 Error Severity Escalation

To simplify error handling or to give certain errors a different priority level, system software may escalate errors to a higher severity level. PAL firmware may permit SAL or the OS to escalate errors. When this feature is enabled and supported, the escalated event signal will be driven out on the processor system bus (ex: BINIT# pin) and will be received by all platform components that are wired to these signals. Table 3-2 shows the different events that can be elevated in severity.

Table 3-2. Machine Check Event Escalation

Processor Detected Machine Check Event	Available Escalation Option
Corrected Machine Check Events	May be promoted to an MCA condition. When this option is chosen, the processor signals a MCA on all CMC conditions. A promoted MCA behaves identically to a local MCA in that it can be further promoted to a BERR# or BINIT# condition.
All MCA Events	May be promoted to a BERR# or BINIT# error condition. When this option is chosen, the processor treats all MCA errors as BERR# or BINIT# error conditions.
Only BERR# Event	May be promoted to BINIT# error condition. When this option is chosen for the detecting processor, the processor treats all BERR# errors as BINIT# error condition.

Although these features are architecturally defined, a particular Itanium architecture implementation may not support all of these capabilities. The PAL_PROC_GET_FEATURES can determine the existence of processor capabilities. The PAL_PROC_SET_FEATURES allows the manipulation of the supported features as well.

§

Platform Error Handling 4

Detecting and reporting platform errors are platform specific. Platform hardware may record error information and return it to the firmware and OS layers, which may have platform-specific error handling capabilities.

4.1 Platform Errors

MCA-enabled platforms may report several different error types, depending upon the platform design. The platform errors can be classified into three categories:

1. Memory errors
2. I/O bus errors
3. OEM-specific errors

Each of the error types will be associated with a unique GUID for identification. OEMs can define their own error type and associated GUID. When these platform errors (BERR#, BINIT#, multi-bit, and Hard Fail bus responses on the system bus) are raised on the system bus, the processors observing the bus condition will indicate external bus errors in their processor error records to the system software (see Section 3.1.3). The platform firmware is responsible for further queries to the platform hardware to identify the source of the error and build an appropriate platform error record.

Since SAL is the platform-specific component of the firmware, it should be able to retrieve any error information from the chipset and possibly correct some errors with a hand-off to the OS. The OS MCA handler in conjunction with the SAL can effectively handle platform-detected errors.

4.1.1 Memory Errors

Errors detected on the external memory subsystem, such as local main memory single-bit or multi-bit errors, will be reported as platform memory errors. The errors detected on any platform-level cache will also fall into this category.

4.1.2 I/O Bus Errors

Errors on I/O interconnects will be reported as platform bus errors. This includes, but is not limited to errors on PCI buses and chipset component interconnect buses. Bus errors on component interconnect buses (e.g. the address and data memory controller buses and associated datapath components) are also reported as platform bus errors.

4.2 Platform Error Correlation

The OS must call SAL_GET_STATE_INFO to retrieve all of the information associated with a machine check event. The processor error record provides the severity of error, which is coupled with the platform error sections returned by SAL. The processor-logged bus error information has an external bus error flag, which is set for errors detected and reported by the platform.

4.3 Platform-corrected Error Signaling

Corrected platform error events are signaled (if OS polling is not used) using two different control paths:

1. Hardware-corrected platform errors
2. Firmware-corrected platform errors

For hardware-corrected platform error, the event notification is sent to the OS with the corrected platform interrupt vector.

The platform may signal a MCA condition through the assertion of external MCA signaling mechanism (BERR#, multi-bit errors and Hard Fail bus responses). If the SAL firmware corrects an error, a corrected platform error event will be signaled to the OS if signaling is enabled. In this case, the SAL will send an IPI to a processor with the corrected platform error vector number.

4.3.1 Scope of Platform Errors

The scope of platform errors depends upon the platform and firmware implementations. Depending upon the platform topology, a single physical platform may consist of multiple processor nodes. A processor node in this context is defined to be a section of the platform that contains a set of processors connected by a bus with its own error event generation and notification ability.

When SAL_GET_STATE_INFO is called for MCA or corrected errors for the platform, SAL returns the error record for the processor node associated with the processor on which the call is made. If a SAL implementation is capable of accessing error information for the entire multi-node system from one processor, it is permitted to aggregate all the platform error sections within one error record.

Returning error information on a processor node basis helps to efficiently manage platform resources for error event notification and error record building when the system has a large number of processors and platform resources.

4.3.2 Handling Corrected Platform Errors

The OS may employ either the CPEI interrupt mechanism or polling to retrieve corrected platform errors. When the OS uses the polling option for the platform corrected error event, it must call SAL_GET_STATE_INFO with argument type of CPE on each of the processor nodes to collate the error information for the entire platform. Note that it is unnecessary to make this call on each processor within a node. If polling is employed, the frequency may be based on past history.

The most common source of corrected platform errors are 1-bit, soft or transient memory errors. This is usually corrected by the 1-bit error correction or double bit error detection feature of the memory controller. The platform hardware would provide the corrected data to the processor but the memory location may still contain erroneous data. Hardware scrubbing will eventually correct the error but this may not be soon enough. It is possible to experience several 1-bit errors from the same location. There may be an abundance of errors if the device driver handling the 1-bit error is located on the faulty location(s) or has the need to access the faulty location(s).

The OS can maintain statistics on the corrected platform errors corrective action since the CPEI interrupt is signaled to the OS first. The only method by which the SAL can perform any pre-processing and thresholding is to define the interrupt signal line for the CPEI to be a PMI interrupt

type in the ACPI tables. This is not a desirable option from a performance perspective as the PMI interrupts are handled at the highest priority level. Further, the resources to maintain statistics may be limited within the SAL data areas.

The OS has several strategies for handling excessive CPEI interrupts:

- Turn off the CPEI interrupt by turning on the “mask” bit within the redirection table entry (RTE) of the I/O SAPIC. This will have the effect of turning off interrupts from all the platform sources that are connected to the RTE. Subsequent to the masking, the OS can use the polling mechanism by invoking the SAL_GET_STATE_INFO procedure. This SAL procedure will clear the error registers within the chipset and enable future logging of errors. The OS can maintain statistics on the location of error, frequency, etc. and vary the polling frequency depending on past history.
- In the event of a HF condition, the code/data from the faulty page should be remapped to a different page and thereafter avoid use of the faulty page. This step is essential so that the 1-bit error does not degrade to a multi-bit error over time. This may be the only way to prevent hard failure memory errors. This solution does not apply if OS invokes the firmware in physical addressing mode and the errant location is part of the firmware image in memory.
- Correct the faulty location permanently by rewriting. The error record associated with the CPEI event would specify the physical address and granularity (address mask) of the faulty memory location(s). The OS should mask the CPEI interrupt for the 1-bit error, read the faulty memory location(s), write it back and execute the **flush cache (*FC*)** instruction to ensure that the location(s) in memory is rewritten. The OS should then invoke SAL procedures to retrieve and clear the error record to ensure that the logging registers within the chipset are freed up for future error logging. The OS may then access the location in uncachable mode to check if the correction was successful. If errors persist, it indicates a stuck-at-fault error.

The SAL may optionally maintain statistics of CPEI interrupts when the OS invokes the SAL_GET_STATE_INFO procedure. For the corrected errors not caused by memory transactions, such as internal buses within the chipset, the SAL may program the chipset registers to avoid corrected error signalling. It may also take platform-specific actions such as system management alerts, enabling alternate paths, etc., if such actions are transparent to the OS execution.

4.4 Platform MCA Signaling

Depending on the severity of an error, a platform has the choice of signaling an error to the processor either in-band to the bus, or out of band. Errors are synchronously reported through multi-bit errors or Hard Fail bus responses, while BERR# or BINIT# are used for asynchronous error reporting.

- *BERR# Pin Assertion:* Since the processors do not reset their internal state, platform errors that are signaled this way may be recoverable. BERR# errors are global events when a platform design connects all the processors’ BERR# pins together in a system.
- *BINIT# Pin Assertion:* Since the processors reset part of their internal state, platform errors signaled this way are not recoverable. BINIT# signaling causes a system-wide MCA event (refer to Section 4.4.1, “Global Signal Routing” on page 4-34 for further details).
- *Forcing a multi-bit Data Error:* Platforms may only use this method on a transaction that requires data return to the processors. On receiving a multi-bit error indicator, for cacheable data, the processor poisons and stores the data in the internal caches. A multi-bit error response is local to the receiving processor.

- *Hard Fail Response Error*: The Hard Fail response is supported by the system bus protocol. On receiving such an error, the processor treats it in the same manner as a multi-bit data error. A Hard Fail error response is local to the receiving processor.

4.4.1 Global Signal Routing

Itanium architecture processors use a multi-layer strategy for error containment at the instruction, process, node, and system levels. PAL firmware performs instruction level error containment. Process level error recovery relies upon the OS to terminate the affected process. At the node and system levels, error containment relies on PAL, SAL, and the OS with the use of the BERR# and BINIT# pins to achieve error containment. The success of this error containment strategy depends on cooperation between the firmware layers and the OS. This section suggests a usage model of the BERR# and BINIT# pins on Itanium architecture-based platforms.

Broadcast of BERR# and BINIT# across processor nodes (see Section 4.3.1, "Scope of Platform Errors" on page 4-32 for the definition of a processor node) is an implementation choice based on the needed platform topology and functionality. Irrespective of the nature of the signal routing, the impact of these platform signal routing choices shall be abstracted from the OS. An example of such abstraction is when a platform does not route the BERR# signal across processor nodes, SAL must perform a SAL rendezvous of the processors on the neighboring processor nodes.

Use of the BERR# Pin: The BERR# pin can be used on an Itanium architecture-based platforms as a way for the platform to signal a recoverable platform MCA. The SAL MCA Handler can specify the error recoverability for platform-asserted BERR# by reporting the appropriate error severity in the error record as being fatal or recoverable. If any platform components lost essential state information due to the assertion of the BERR# pin, then SAL must report the error as fatal.

Current Itanium processors drive the BERR# pin for errors that are not recoverable but are contained within the processor. The processor drives this pin to notify other processors that there is an unrecoverable error and to get the other processors to stop their currently executing programs, in order to reduce the chance of one of these processors noticing the same error (this would cause the processor to assert the BINIT#). This increases the chance of allowing the firmware and OS to get a good error log stored before having to reboot the system. For very large multi-node systems, platforms may not want to tie the BERR# pins together across processor nodes, but can achieve a similar global event notification by using the rendezvous mechanism to coordinate error handling.

Use of the BINIT# Pin: The BINIT# pin is used for system level error containment. The processor only asserts this pin for fatal errors that may cause loss of error containment. These pins shall be tied together in a multi-node system. BINIT# assertion is a system wide MCA event.

To be consistent with the Machine Check Architecture, it is recommended that the platform hardware generate a BERR#, multi-bit errors, data poisoning, or Hard Failure bus responses for recoverable errors and a BINIT# for fatal errors. Since BERR# assertion allows the MCA handlers to make forward progress, some platforms may choose to report containable fatal errors through this means rather than asserting BINIT#.

4.4.2 Error Escalation

In addition to the processor error masking capabilities, the platform hardware may also provide an implementation-specific way of masking the platform to external interrupts for error severity escalation.

Platform errors that are visible to the processor can be escalated on a per processor basis by setting the processor configuration bits as shown in Table 3-2.

The chipset may also have capabilities to escalate all platform errors to BERR# (potentially recoverable) or BINIT# errors (non-recoverable).

§

Error Records 5

This chapter discusses the classification of machine check error records. It also provides guidelines on how the error records should be handled by the firmware and OS.

5.1 Error Record Overview

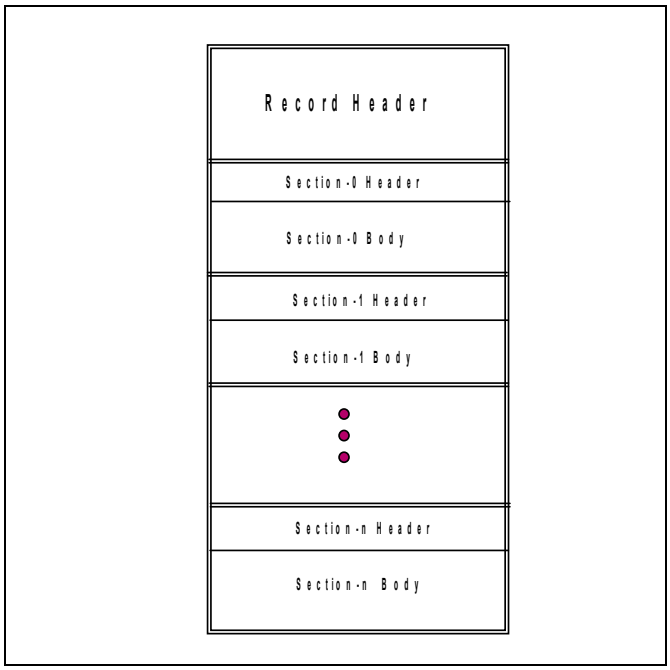
The error records captured on a system are associated with error events. Corresponding to each event is one error record with processor and/or platform error sections. Error records are important for error tracking and recovery. SAL (in conjunction with PAL) shall maintain the error record information of all uncorrected errors (MCAs) in a non-volatile local memory (NVM) area. Storing the MCA information in the NVM permits this error information to be retrieved after a system reboot. The log for corrected error records (CMC and CPE) is not required to be stored in the NVM.

5.2 Error Record Structure

The *Intel® Itanium® Processor Family System Abstraction Layer Specification* defines an error record structure, which is used to return error information to the OS. The error record structure may consist of multiple sections for each system component. The format of the error record for an event is as shown in Figure 5-1.

Each of the sections of the error record has an associated globally unique ID (GUID) to identify the section type as being processor, platform bus, or other OEM-specific type.

Figure 5-1. Error Record Format



5.2.1 Required and Optional Error Sections

The SAL implementation shall provide all the information significant for logging and identification of the field replaceable unit and recovery. However, the particular SAL implementation may not implement all the sections in an error record. The following is a list of error sections that must be supported by all SAL implementations:

- Processor Device Error Info section
- Platform Memory Device Error Info section
- PCI Bus Error Info section
- Platform PCI Component Error Info section

The following is a list of error sections that may *optionally* be present in SAL implementations. Three of the following are OEM-specific and not critical for OS error handling:

- Platform SEL Device Error Info section
- Platform SMBIOS Device Error Info section
- Platform Specific Error Info section

Within an error section, not all fields may be present. The *Intel® Itanium® Processor Family System Abstraction Layer Specification* describes valid bits to indicate the presence of fields within a section. Some of the fields may not be usable by the OS as these may be OEM-specific information (e.g. serial number, firmware build number on an embedded controller, etc.). These may be usable by the OEM_OS_MCA layer and by OEM system management software.

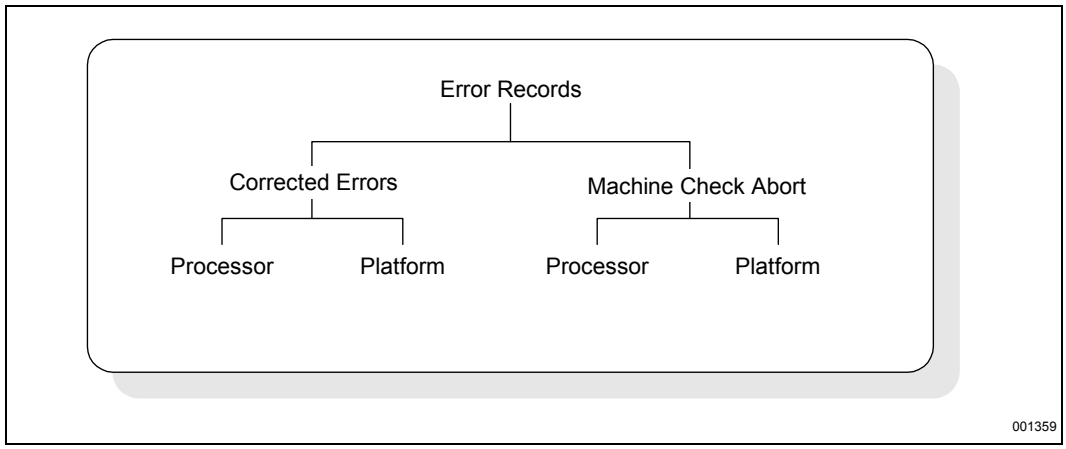
The Platform Memory Device Error section shall contain valid information for the physical address of the error and the address mask (MEM_PHYSICAL_ADDR and the MEM_PHYSICAL_ADDR_MASK fields). Refer to the *Intel® Itanium® Processor Family System Abstraction Layer Specification* for details.

5.3 Error Record Categories

All uncorrected and corrected (CMC and CPE) processor/platform errors events have associated error records. Thus, the error records are classified into (refer to Figure 5-2):

1. MCA Records
2. CMC and CPE Records

Figure 5-2. Error Record Tree



5.3.1 MCA Record

In general, each MCA event in the system can have no more than one MCA error record per processor and one error record for the entire physical platform[1] at any given point in time. The only exception is if MCA events occur during the booting process and SAL logs the unconsumed MCA to NVM. While the software is handling the current MCA, any further MCAs are held pending, so no new MCA records are built. Error records for subsequent MCA events will be built and made available after the OS completes the following sequence:

1. Retrieve the previous record.
2. Complete the MCA handling.
3. Initiate an explicit call to SAL (SAL_CLEAR_STATE_INFO) to clear the MCA records. This call also marks the corresponding record in the NVM as consumed.
4. Unmask MCA interrupt detection on the processor.

Note: During an MCA event, the error record returned by SAL_GET_STATE_INFO may contain valid sections for processor and/or platform errors. The different sections that are returned by SAL during this event depend on the event type and the SAL implementation. Refer to the *Intel® Itanium® Processor Family System Abstraction Layer Specification* for a description of the error records and the APIs for error record retrieval.

5.3.2 CMC and CPE Records

Each processor or physical platform could have multiple valid corrected machine check or corrected platform error records. The maximum number of these records present in a system depends on the SAL implementation and the storage space available on the system. There is no requirement for these records to be logged into NVM. The SAL may use an implementation-specific error record replacement algorithm for overflow situations. The OS needs to make an explicit call to the SAL procedure SAL_CLEAR_STATE_INFO to clear the CMC and CPE records in order to free up the memory resources that may be used for future records.

1. Please refer to the *SAL Specification* for more details on the scope of platform corrected errors. Platform error section will not be present if the MCA is internal to the processor.

Note: During a corrected error event, SAL returns error records consisting of appropriate error sections for the event type, namely a processor section for the CMC and a platform section for the CPE. In some situations, when platform errors are reported by the platform using synchronous MCA signaling (2-bit errors or Hard Failure bus responses), the SAL may correct the error and signal a CPE event.

5.4 Error Record Management

The management of the error records (MCA, CMC, and CPE) depends upon the SAL implementation. Records may one of two states:

1. Consumed
2. Not Consumed

Consumed error records have been read by the OS and cleared using SAL_CLEAR_STATE_INFO. An OS typically clears a record after it has been written to the OS event log on the disk. In case of system reboot without an OS's explicit clear (not consumed), the unconsumed records would still be available across system reboots for use by the OS. SAL firmware can maintain a consummation state flag for each error record in the NVM for its management.

Note: SAL may choose to implement separation of "Consumed" and "Not Consumed" error records, however it is not architecturally required. A platform implementation with NVM support could keep all OS-consumed error records in the NVM for utility software. If this implementation approach is taken, all error records can be made available to field service personnel through the utility software.

5.4.1 Corrected Error Event Record

In response to a CMC/CPE condition, SAL builds and maintains the error record for OS retrieval. An OS might choose different CMC/CPE event notification types during OS boot by configuring the firmware through SAL_MC_SET_PARAMS. In response to the CMC/CPE event (CMCI or CPEI), firmware error record management routines could follow the sequence described below:

1. The OS calls SAL (interrupts enabled) and gets the corrected event error record.
2. The OS clears corrected event error record (as an indication of the end of event handling) by clearing SAL_CLEAR_STATE_INFO, the FW could do the following:
   - SAL clears the error record from memory.
   - SAL may *optionally* write the error record to NVM as "Consumed" by the OS. Most SAL implementations may not perform this step as the equivalent information would be available from the OS resources.
   - SAL may optionally perform garbage collection on NVM error records during the SAL_CLEAR_STATE_INFO call. The NVM garbage collection latency would not have any performance impact on the OS since this call is made with interrupts enabled.

5.4.2 MCA Event Error Record

In response to an MCA condition, SAL builds and maintains the error record for OS retrieval. During the OS_MCA handling or at a subsequent point, the OS would get the error record for the current MCA from the firmware. The error record management by firmware could follow the sequence as outlined below.

1. SAL_CHECK hands off to OS_MCA after the following:
   - — SAL builds the error records.
   - — SAL writes the error record to NVM.
   - — SAL marks the error record in NVM as “Not Consumed” by the OS.
2. After OS_MCA hand-off, OS calls SAL and gets the MCA error record.
3. The OS clears MCA error record (as an indication of the end of MCA handling) through SAL_CLEAR_STATE_INFO. SAL would then do the following:
   - — SAL clears the error record from memory.
   - — SAL marks the error record in NVM as “Consumed” by the OS.

5.4.3 Error Records Across Reboots

Unrecoverable error events appear to the OS as MCA conditions. In some cases, the OS might not be able to write error records into its persistent storage, so the SAL error record remains unconsumed. In these situations, the OS would reboot the system clearing the error record in memory. To take this situation into account, the firmware needs to maintain the error records across system reboots. The following is a typical sequence for the OS to obtain the error record across reboots.

1. On reboot, the OS requests the Unconsumed MCA Event error records by calling the SAL_GET_STATE_INFO procedure. The SAL will do the following:
   - — SAL will provide the error record if it exists in memory or (since the OS has not cleared the error record yet) from the NVM.
2. The OS clears the error record through the SAL_CLEAR_STATE_INFO procedure. SAL marks the error record in NVM as “Consumed” by the OS.
   - — If the OS fails to clear the log for the previous boot session before another MCA surfaces, SAL may choose to overwrite the unconsumed NVM log, if there is not space for another record. The SAL implementation may additionally escalate the error severity in the Record header when the error information is subsequently provided to the OS.

5.4.4 Multiple Error Records

As outlined earlier, it is possible for the platform to have multiple error records stored in the system NVM that are not yet consumed by the OS. SAL must choose how to store the errors records in the NVM and in what order to return them. Here are some guidelines for typical implementations:

Corrected Error Records: The OS uses the corrected error records from SAL_GET_STATE_INFO for logging purposes and error records are retrieved and cleared by the OS one at a time using a first-in-first-out (FIFO) method.

MCA Error Records: An MCA record returned during an MCA hand-off to the OS should always return the current record, because the OS may use it for recovery. If multiple unconsumed MCA records are present in the NVM that do not pertain to the current MCA event, they may be used by the OS for logging and hence the order can be SAL implementation specific.

§

Error Injection A

Machine Check Aborts (MCA) can be intentionally generated using special hardware that creates MCAs using memory errors, PCI bus errors, etc. On the Itanium architecture implementation, a number of errors may also be injected through software. This capability can be used by diagnostic software for testing and to verify the PAL, the SAL and the OS error handling flow. Some examples of error injection are described below. These are by no means exhaustive.

These error injection capabilities can be used in conjunction with the MC Expected (MCE) state, programmed through the PAL call. When this state is turned on, PAL machine check handler will not attempt an error recovery, but will hand off the machine check directly to the SAL.

A.1 Platform I/O Errors

Some examples of CMCI and MCA injection using the platform hardware are given below. These examples may not be universally applicable to Itanium processor implementations as platforms may differ in their choice of components and even while using the same components, their configuration settings and platform addresses may vary.

CMCI Generation: Chipsets such as the Intel® 460GX may be programmed to generate a 1-bit error on the system bus (e.g. using the ERRMSKF register of the 460GX chipset). The processor would automatically correct the 1-bit error and signal a hardware-corrected CMCI to the OS.

BERR# Generation: The PID interrupt controller provides a register in the PCI configuration space that forces the SERR# signal on the PCI bus. Other chipsets on the platform may transform the SERR# indication on the PCI bus to a BERR# signal on the processor system bus.

Other platform hardware can be programmed to generate external asynchronous BERR# and BINIT# signal assertion, or generate synchronous Hard Fail response and multi-bit errors on any bus transactions.

A.2 Memory Errors

CPEI and Local MCA Generation: Many memory controllers provide the option to seed 1-bit and multi bit errors into memory (e.g. ERRMSK*n* register on the 460GX chipset). The chipset may be set to the mode for seeding errors and some memory locations may be written with bad data. When the data is accessed subsequently, the chipset would generate a CPEI for 1-bit corrected errors and a local MCA for multi-bit uncorrected errors. The OS must be careful to limit memory traffic while seeding errors, otherwise the seeding may not be limited to the intended memory locations.

§

Pseudocode – OS_MCA B

The following pseudocode is provided to show the steps an OS would take to try and recover from a processor cache error. This pseudocode is not meant to be an exhaustive definition of what an OS needs to do for all machine check handling, but rather a starting point for OS designers.

```
/*====================================================================*/
/* Definitions - These are provided to attempt to make the pseudo     */
/*               code easier to read and are not meant to be real     */
/*               definitions that can be used.                        */
/*====================================================================*/
```

Note: Processor State Parameter is located in PSP=r18 at hand off from SAL to the OS_MCA handler. Define some PSP bit fields.

```
/* Processor State Parameter bit field definitions */
define TLB_Error   = ProcessorStatParameter[60]

/* SAL Record Header Error Log Definitions */
#define Record_ID_Offset     = 0
#define Err_Severity_Offset = 10
#define Recoverable = 0
#define Fatal       = 1
#define Corrected = 2
#define Record_Length_Offset = 12
#define Record_Header_Length = 40

/* SAL Section Header Error Log Definitions */

#define GUID_Offset = 0
#define Section_Length_Offset = 20
#define Processor_GUID = E429FAF1-3CB7-11D4-BCA70080C73C8881
#define Section_Header_Length = 24

/* SAL Processor Error Record Definitions */

#define Validation_Bit_Structure
    Proc_Error_Map_Valid       = bit 0
    Cache_Check_Valid          = bits [7:4]
    TLB_Check_Valid            = bits [11:8]
    Bus_Check_Valid            = bits [15:12]
    Reg_File_Check_Valid       = bits [19:16]
    MS_Check_Valid             = bits [23:20]

#define Error_Validation_Bit_Length = 8
#define Check_Info_Valid_Bit           = bit 0
#define Target_Address_Valid_Bit       = bit 3
#define Precise_IP_Valid_Bit           = bit 4

#define Check_Info_Offset          = 0
#define Target_Address_Offset      = 24
```

```
#define Precise_IP_Offset           = 32

/* Cache Check Info Bit definitions */

#define PrecisePrivLevel       = bits [57:56]
#define PrecisePrivLevel_Valid = bits 58

/*=========================BEGIN=======================================*/
/* OS Machine Check Initialization                                     */
/*=====================================================================*/
OS_MCA_Initialization()
{
/* this code is executed once by OS during boot, on the Bootstrap processor.
Register OS_MCA Interrupt parameters by calling SAL_MC_SET_PARAMS */

    Install OS_Rendez_Interrupt_Handler
    Install OS_Rendez_WakeUp_Interrupt_Handler /* ISR clean up wrapper */
    Register_Rendez Interrupt_Type&Vector;
    Register_WakeUpInterrupt _Type&Vector;
    Specify the options during MCA processing (rendezvous always, MCA to BINIT
escalation);
    Register_CorrectedPlatformErrorInterrupt_Vector with SAL;
    Program the I/O SAPIC for CPEI interrupt;
    Initialize_CMC_Vector_Masking;

/* Register OS_MCA Entry Point parameters by calling SAL_SET_VECTORS */

    Register_OS_MCA_EntryPoint;
    Register_OS_INIT_EntryPoint;
}
/*============================END======================================*/

/*=========================BEGIN=======================================*/
/* OS Machine Check Rendez Interrupt Handler                           */
/*=====================================================================*/
OS_Rendez_Interrupt_Handler()
{
    /* go to spin loop */
    Mask_All_Interrupts;
    Set Flag to indicate Rendezvous occurrence (with value based on CPUID);
    Call SAL_MC_RENDEZ();

    /* clean-up after wakeup from exit */
    Clear Flag to indicate Rendezvous occurrence;
    Enable_All_Interrupts;

    /* return from interruption */
    return;
}
/*============================END======================================*/

/*=========================BEGIN=======================================*/
/* OS Corrected Error Interrupt Handler (processor and platform)       */
/*=====================================================================*/
OS_Corrected_Error_Interrupt_Handler()
{
/* handler for corrected machine check intr.*/
```

```
    /* get error log */
    if(ProcessorCorrectedError)
        Sal_Get_State_Info(processor);
    else
        Sal_Get_State_Info(platform)
```

Note: If saving of the error record is to disk or the OS event log, then this is core OS functionality.

```
    /* Save log of MCA */
    Save_Error_Log();

    /* now we can clear the errors */
    if(ProcessorCorrectedError)
        Call Sal_Clear_State_Info(processor);
    else
        Call Sal_Clear_State_Info(platform);

/* return from interruption */
        return;
}
/*==========================END=====================================*/

/*==========================BEGIN=====================================*/
/* OS Core Machine Check Handler                                      */
/*====================================================================*/
OS_MCA_Handler()
{
/* handler for uncorrected machine check event */
    Save_Processor_State();

    if(ErrorType!=Processor TLB)
        SwitchToVirtualMode();
    else
        StayInPhysicalMode();

    /* Assuming that the OS can call SAL in physical mode to get info */
    SAL_GET_STATE_INFO(MCA);

    /* check for error */
    if(ErrorType==processor)
    {
        if(ErrorType==processor TLB)
            // reset the system and get the error record at reboot
            SystemReset() or ReturnToSAL(failure);
        else
            ErrorCorrectedStatus=OsProcessorMca();
    }
    If(ErrorType==Platform)
    {
        Elect a Monarch processor;
        ErrorCorrectedStatus|=OsPlatformMca();
    }
```

Note: If the error is not corrected, OS may want to reboot the machine and can do it by returning to SAL with a failure return result.

```
    If(ErrorCorrectedStatus==failure)
        branch=ReturnToSAL_CHECK
```

Note: Errors are corrected, so try to wake up processors which are in Rendezvous. OS data structures should indicate which processors have rendezvoused.

```
    /* completed error handling */
    if(ErrorCorrectedStatus==success && InRendezvous()==true)
        WakeUpApplicationProcessorsFromRendezvous();
```

Note: If saving of the error record is to disk or the OS event log, then this is core OS functionality.

```
    /* as a last thing */
    Save_Error_Log();
```

Note: This is a very important step, as this clears the error record and also indicates the end of machine check handling by the OS. SAL uses this to clear any state information it may have related to which processors are in the MCA and any state of earlier rendezvous.

```
    Call Sal_Clear_State_Info(MCA);

ReturnToSAL:
    /* return from interruption on all the processors that entered OS_MCA*/
    Switch To Physical Mode();
    Restore_Processor_State();

    /* return to SAL CHECK, SAL would do a reset if OS fails to correct*/
    return(ErrorCorrectedStatus)
}
/*============================END=====================================*/

/*=========================BEGIN=====================================*/
/* OS Platform Machine Check Handler                                  */
/*====================================================================*/
OsPlatformMca()
{
    ErrorCorrectedStatus=True;

    /* check if the error is corrected by PAL or SAL */
    If(ErrorRecord.Severity==not corrected)
        /* call sub-routine to try and correct the Platform MCA */
        ErrorCorrectedStatus=Correctable_Platform_MCA(platform_error_type);

    return(ErrorCorrectedStatus);
}
/*============================END=====================================*/

/*=========================BEGIN=====================================*/
/* OS Processor Machine Check Handler                                 */
/*====================================================================*/
OsProcessorMca()
{
    ErrorCorrectedStatus=True;

    /* check if the error is corrected by Firmware */
    If(ErrorRecord.Severity==not corrected)
```

```
        ErrorCorrectedStatus=TryProcessorErrorCorrection();

    Return(ErrorCorrectedStatus);
}

/*=========================END=====================================*/

/*=========================BEGIN=====================================*/
/* Try Individual Processor Error Correction                          */
/*====================================================================*/
```

Note: Now the OS has the data logs. Start parsing the log retrieved from SAL. The sub-routine Read_OS_Error_Log will read data from the error log copied from SAL. An offset is passed to identify the data being read and the base pointer is assumed to be known by the Read_OS_Error_Log sub-routine just to simplify the pseudo-code.

```
TryProcessorErrorCorrection()
{
    /* extract appropriate fields from the record header */
    Record_ID = Read_OS_Error_Log(Record_ID_Offset);
    Severity = Read_OS_Error_Log(Err_Severity_Offset);
```

Note: It is unlikely that the OS can write to persistent storage in physical mode. If it is possible, the OS should do so. If it is not, the SAL firmware should still have a copy of the error log stored to NVM that will be persistent across resets.

```
    if (Severity == Fatal)
        SystemReset() or return(failure);

    if (Severity == Corrected)
        return(ErrorCorrectedStatus=True);
```

Note: These errors may be recoverable by the OS depending on the OS capability and the information logged by the processor. Call the sub-routine, OS_MCA_Recovery_Code and on return set up a min-state save area to return to a context of choice. The PAL_MC_RESUME call made by SAL allows the OS to return to the interrupted context, which includes enabling of all pending MCAs.

```
    if (Severity == Recoverable)
    {
        ErrorCorrectedStatus=OS_MCA_Recovery();
        Set_Up_A_Min_State_For_OS_MCA_Recovery(my_minstate);
    }
    return(ErrorCorrectedStatus);

} /* End of TryProcessorErrorCorrection Handler */
/*=========================END=====================================*/

/*=========================BEGIN=====================================*/
/* OS_MCA Recovery Code                                               */
/*====================================================================*/
```

Note: At this point the OS is running with address translations enabled. This is needed otherwise the OS would not be able to access all of its data structures needed to analyze if the error is recoverable or not.

```
OS_MCA_Recovery()
{
    /* Set up by default that the errors are not corrected */
    CorrectedErrorStatus = CorrectedCacheErr = CorrectedTlbErr =
    CorrectedBusErr = CorrectedRegFileErr = CorrectedUarchErr = 0;


    /* Start parsing the error log */
    RecordLength = Read_OS_Error_Log(Record_Length_Offset);
    Section_Header_Offset = OS_Error_Log_Pointer + Record_Header_Length;

    /* Find the processor error log data */
    Processor_Error_Log_Found = 0;

    /* traverse the error record structure to find processor section */
    while (Processor_Error_Log_Found == 0)
    {
        SectionGUID = Read_OS_Error_Log(Section_Header_Offset + GUID_Offset);
        SectionLength = Read_OS_Error_Log(Section_Header_Offset +
                                              Section_Length_Offset);
        if (SectionGUID == Processor_GUID)
            Processor_Error_Log_Found = 1;

        Section_Body_Pointer = Section_Header_Offset + Section_Header_Length;
        Section_Header_Offset = Section_Header_Offset + SectionLength;

        if (Section_Header_Offset >= RecordLength)
            InternalError();     /* Expecting a processor log */
    }
```

Note: Start parsing the processor error log. Section_Body_Pointer was set up to point to the first offset of the processor error log in the while loop above. Check the valid bits to see which part of the structure has valid info. The Read_OS_Error_Log sub-routine is assumed to know the initial pointer and just an offset is passed. This was done to allow the pseudo-code to be more readable.

```
    Proc_Valid_Bits = Read_OS_Error_Log(Section_Body_Pointer);
    Section_Body_Pointer = Section_Body_Pointer + Validation_Bit_Length;

    /* Read the Processor Error Map if the valid bit is set. */
    if (Proc_Valid_Bits[Proc_Error_Map_Valid] == 1)
        Proc_Error_Map = Read_OS_Error_Log(Section_Body_Pointer);

    /* Extract how many errors are valid in the error log and determine which type
*/
    Cache_Check_Errs = Proc_Valid_Bits[Cache_Check_Valid];
    TLB_Check_Errs = Proc_Valid_Bits[TLB_Check_Valid];
    Bus_Check_Errs = Proc_Valid_Bits[Bus_Check_Valid];
    Reg_File_Errs = Proc_Valid_Bits[Reg_File_Check_Valid];
    Uarch_Errs = Proc_Valid_Bits[MS_Check_Valid];

    /* These sub-routines will return an indication of if the error can be
        corrected by killing the affected processes. */
```

```
    if (Cache_Check_Errs != 0)
    {
        /* If uncorrected cache errors are reported, the OS should attempt not to */
        /* use the cache. This would require the OS to not caching any of the */
        /* code or data used during recovery. */
        /*                                    */
        /* Check to see if one or multiple cache errors occurred */
        if (Cache_Check_Errs == 1)
            CorrectedCacheErr = Handle_Single_Cache_Error(Section_Body_Pointer);
        else
            CorrectedCacheErr = Handle_Multiple_Cache_Errors(Section_Body_Pointer);
    }

    if (TLB_Check_Errs != 0)
    {
        /* Check to see if one or multiple TLB errors occurred */
        if (TLB_Check_Errs == 1)
            CorrectedTlbErr = Handle_Single_TLB_Error(Section_Body_Pointer);
        else
            CorrectedTlbErr = Handle_Multiple_TLB_Errors(Section_Body_Pointer);
    }

    if (Bus_Check_Errs != 0)
    {
        /* Check to see if one or multiple Bus errors occurred */
        if (Bus_Check_Errs == 1)
            CorrectedBusErr = Handle_Single_Bus_Error(Section_Body_Pointer);
        else
            CorrectedBusErr = Handle_Multiple_Bus_Errors(Section_Body_Pointer);
    }

    if (Reg_File_Errs != 0)
    {
        /* Check to see if one or multiple Register file errors occurred */
        if (Reg_File_Errs == 1)
            CorrectedRegFileErr = Handle_Single_Reg_File_Error(Section_Body_Pointer);
        else
            CorrectedRegFileErr = Handle_Multiple_Reg_File_Errors(Section_Body_Pointer);
    }

    if (Uarch_Errs != 0)
    {
        /* Check to see if one or multiple Uarch file errors occurred */
        if (Uarch_Errs == 1)
            CorrectedUarch_Err = Handle_Single_Uarch_Error(Section_Body_Pointer);
        else
            CorrectedUarch_Err = Handle_Multiple_Uarch_Errors(Section_Body_Pointer);
    }

    CorrectedErrorStatus = CorrectedCacheErr | CorrectedTlbErr | CorrectedBusErr |
                           CorrectedRegFileErr | CorrectedUarch_Err;

    return(CorrecteErrorStatus);
}   /* end OS_MCA_Recovery_Code */
/*==========================END=====================================*/
```

§